designing state machines

Daniel Jackson

February 6, 2008
why state machines?
why models?

- biggest challenge: bridging the problem-code gap
- models capture essence -- good for exploring problems
- models hide details -- good for exploring solutions
- same concepts in problem and solution models
state machine models

standard model

• states + actions + transitions
• good for ‘control-intensive systems’
• wide use in industry (eg, medical devices, cars, network protocols)

variants

• timed: allow analysis of real-time deadlines
• stochastic: attach probabilities to transitions
• hybrid: handle continuous variables too

but I’m not interested in embedded systems...

• every system has a control aspect
• user interaction usually has strong control aspect
state machines more generally

not just for small machines
• useful even if too big to draw
• state machine is a concept, multiple ways to express
• state machines don’t have to be finite

conceptual models
• state machines are a great basis for semantics
• conceptual model that underlies reasoning about abstract types
• good way to understand the Java language
designing a midi piano
rough goals

functionality

• play notes on computer keyboard
• sustain by holding key down
• record and playback

resources

• Java midi API

quality goals

• initial prototype needed
• not super-dependable
getting going

“description before invention”

context

• piano depends on keyboard, midi API
• this is a dependence diagram -- more later

need to model

• keyboard input: press and release
• MIDI interface: commands
a single key

modelling the user

- two states, KEYUP (initial) and KEYDN
- two input event classes
  - press: the user presses down on the key
  - release: the user releases the key

the keyboard driver

- add output event classes
  - pr: the driver emits event reporting press
  - rel: the driver emits event reporting release

key repeat

- model time with explicit tick event
- each tick produces another pr output
two keys

what happens with two keys?
- product of states: \(2 \times 2 = 4\)

another way to show
- dotted line separates parallel machines

semantically equivalent
- machines have same \textit{traces}

but are they really independent?
- how does key repeat behave?
two keys with repeating

exercise for the reader

• hold down one key and wait for repeat
• now press another: what happened?
• choose a diagram from previous slide
• embellish it with ticks and the events they generate
what’s a state machine?

a state machine consists of

ˌ a set of states
  State = \{ KEYUP, KEYDN \}

ˌ a set of initial states
  Init = \{ KEYUP \}

ˌ input and output event sets
  Event = \{ press, release, pr, rel \}
  InputEvent = \{ press, release \}
  OutputEvent = \{ pr, rel \}

ˌ a transition relation
  trans ⊆ State × Label × State =
    \{ (KEYUP, <press,pr>, KEYDN), (KEYDN, <release,rel>, KEYUP) \}
  Label is sequence of: <input>, <output>, or <input, output>
semantics

what does it mean?
• how would you know if it’s right?

meaning of machine is trace set
• all possible event sequences
  \[
  \text{traces} \subseteq \text{Event}^* \\
  \text{traces} = \\
  \{ <>, <\text{press}>, <\text{press, pr}>, <\text{press, pr, release}>, \\
  <\text{press, pr, release, rel}>, \\
  <\text{press, pr, release, rel, press}>, \ldots \} \\
  \]

a subtle issue
• what if event cannot be added to trace?
  • two possibilities: ignored or blocked
• usually we’ll assume ignore semantics
modelling advice

grounding in reality

• select events first and thoughtfully
• give explicit designations saying what they mean
• must be atomic and recognizable

bad smells

• events not designated, event classes not really disjoint
• no initial state marked
• some states or transitions have no labels

not a control flow graph!

• no behaviour inside states
• no ‘decision edges’
modelling the midi API
piano design: first step

events

- inputs: k-pr, k-rel for all keys k
- outputs: k-begin, k-end, init
recording modes

introduce

• events
  R-pr: press R key to toggle recording

• modes
  !REC: recording off
  REC: recording on

consider traces

• how about this?
  <k-pr, R-pr, k-rel, R-pr>

in more serious development

• would design machine to rule this out
• but we’ll ignore it for now
playback

let's add a playback mode

- use P key for playing back
  
P-pr: user presses P to start playback

- introduce *internal* event
  
done: playback is completed

mixing manual play and playback

- can k-pr, k-rel occur during playback?

playback before record?

- what happens if you do this?
  
  <P-pr>
record during playback

can you record during playback?
  · very useful, but much trickier

traces to consider
  · how about this?
    $<\text{R-pr, P-pr, R-pr, done}>$

options
  · ignore R-pr until done (inconvenient)
  · abort playback somehow (complicated)
  · allow and handle effect on recording (best?)
useful notation

same machine

• as on previous slide
• but some extra notation
• from Harel's statecharts

new elements

• ‘superstates’: hierarchy of states
• transitions out of superstates
• transitions into superstates
  with initialization and history
asynchronous playback

idea
• playback generates k-pr and k-rel events
• queue these events on same queue as incoming k-pr, k-rel
arrows are data/event flows

- queue merges events non-deterministically
- midi piano sees just one stream of events
- processes events sequentially
- significance will become clearer later (when we do concurrency)
more on recording state

what would happen if

• playback is pressed during record
• generated events get played
• and added to recording
• so they get played back again...

oops!

• keep two recordings
  
  **current:** holds events being recorded  
  **last:** holds events being played back

• how are these updated?

• need a richer state machine syntax
a textual state machine

```
state
  List<Event> last, current = <>;
boolean REC = false;

op R-pr
  when true
  do
    if (REC) last = current else current = <>;
    REC = !REC
```

elements

- state components with initializations
- operations each with preconditions and postcondition
- precondition (‘when’) says when event can occur
- postcondition (‘do’) says how state is updated when it occurs
final state machine

state
List<Event> last, current = <>;
boolean REC = false;

op R-pr
  do if (REC) last = current else current = <>; REC = !REC

op pr(k) // note how I made k a parameter to generalize over all keys
  do if (REC) current = current ^ <k>

© Daniel Jackson 2008
summary
what did we do?

“description before invention”

• started by modelling keyboard driver

state machine models

• simple but powerful
• focus on the bit the matters, not exhaustive detail
• many small models, not one huge one
• simple syntax, can mix graphical and textual

trickier notions were

• parallel submachines
• feeding playback events back into queue
lecture exercise

by next lecture (Monday) in your LNBs

construct state machine models for two of these

- the mouse on your computer
- the course registration process at MIT
- the call waiting protocol on your phone

using textual syntax

- construct machine with inputs
  
  k-pr, k-rel

- that on R-pr toggles state component
  
  boolean isRecording

- but ignores R-pr when any key is down
feedback

please help us improve the course

‣ write a few sentences in your LNB after each lecture

questions

‣ what was the most important idea in today's lecture?
‣ what did you find confusing?
‣ what examples or exercises helped or didn't help you?